(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 425 421 A2**

# (12) EUROPEAN PATENT APPLICATION

(21) Application number: **90480156.0**

(51) Int. Cl.[5]: **G06F 9/44**, G06F 15/40

(22) Date of filing: **09.10.90**

(30) Priority: **23.10.89 US 425813**

(43) Date of publication of application:
**02.05.91 Bulletin 91/18**

(84) Designated Contracting States:
**DE FR GB**

(71) Applicant: **International Business Machines Corporation**
**Old Orchard Road**
**Armonk, N.Y. 10504(US)**

(72) Inventor: **Shackelford, Floyd Wayne**
**3510 Hanover Drive**
**Buford, GA 30518(US)**

(74) Representative: **Bonneau, Gérard**
**Compagnie IBM France Département de Propriété Industrielle**
**F-06610 La Gaude(FR)**

## (54) Process and apparatus for manipulating a boundless data stream in an object oriented programming system.

(57) A process and apparatus for manipulating boundless data streams in an object oriented programming system (11) provides a stream class of objects which includes as attributes an ordered list of object references to selected ones of the data objects stored in the data storage device (14). Methods for manipulating the object include move to first. move to last. move to next and move to previous which provide a bi-directional data stream. The data appears to the user as though it resides entirely in memory, even though it does not. The stream class implements a "sliding window" in an object oriented programming system which permits manipulation of any number of lists of virtually unlimited size when remaining within the physical limitations of finite storage.

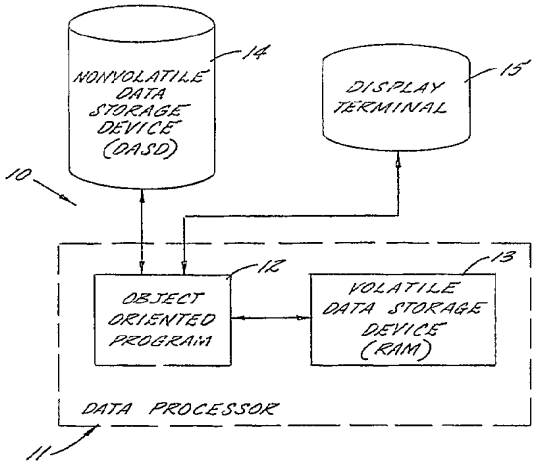



Fig. 4.

EP 0 425 421 A2

# PROCESS AND APPARATUS FOR MANIPULATING A BOUNDLESS DATA STREAM IN AN OBJECT ORIENTED PROGRAMMING SYSTEM

This invention relates to object oriented programming systems and more particularly to a method and apparatus for manipulating a boundless data stream in an object oriented programming system.

Object Oriented Programming systems and processes have been the subject of much investigation and interest in state of the art data processing environments. Object Oriented Programming is a computer program packaging technique which provides reusable and easily expandable programs. In contrast with known functional programming techniques which are not easily adaptable to new functional requirements and new types of data, object oriented programs are reusable and expandable as new requirements arise. With the ever increasing complexity of computer based systems, object oriented programming has received increased attention and investigation.

In an object oriented programming system, the primary focus is on data, rather than functions. Object oriented programming systems are composed of a large number of "objects". An object is a data structure and a set of operations or functions that can access that data structure. The data structure may be represented as a "frame". The frame has many "slots", each of which contains an "attribute" of the data in the slot. The attribute may be a primitive (i.e. an integer or string) or an Object Reference which is a pointer to the object's instance or instances (defined below). Each operation (function) that can access the data structure is called a "method".

Figure 1 illustrates a schematic representation of an object in which a frame is encapsulated within its methods. Figure 2 illustrates an example of an object, in which the data structure relates to employee data, and a number of methods surround this data structure. One method, for example, obtains the age of an employee. Each defined object will usually be manifested in a number of instances. Each instance contains the particular data structure for a particular example of the object. For example, an object for individual employee named Joyce Smith is an instance of the "employee" object.

Object oriented programming systems provide two primary characteristics which allow flexible and reusable programs to be developed. These characteristics are referred to as "encapsulation" and "inheritance". As may be seen from Figure 1, the frame (data set) is encapsulated by its methods (functions). A wall of code has been placed around each piece of data. All access to the frame is handled by the surrounding methods. Data independence is thereby provided because an object's data structure is accessed only by its methods. Only the associated methods know the internal data structure. This ensures data integrity.

The "inheritance" property of object oriented programming systems allows previously written programs to be broadened by creating new superclasses and subclasses of objects. New objects are described by how they differ from preexisting objects so that entirely new programs need not be written to handle new types of data or functions.

Figure 3 illustrates the inheritance property. For ease of illustration, the objects are illustrated as rectangles rather than as circles, with the object name at the top of a rectangle, the frame below the object name and the methods below the frame. Referring to Figure 3, three object classes are illustrated for "salesperson", "employee" and "person", where a salesperson is a "kind of" employee, which is a "kind of" person. In other words, salesperson is a subclass of employee and employee is the superclass of salesperson. Similarly, employee is the subclass of person and person is the superclass of employee. Each class shown includes three instances. B. Soutter, W. Tipp and B. G. Blue are salespersons. B. Abraham, K. Yates and R. Moore are employees. J. McEnro, R. Nader and R. Reagan are persons. In other words, an instance is related to its class by an "is a" relation.

Each subclass "inherits" the frame and methods of its superclass. Thus, for example, a salesperson frame inherits age and hire date objects from the employee superclass as well as print and promote methods. Salesperson also includes a unique quota attribute and a pay commission method. Each instance can access all methods and frames of its superclass, so that, for example, B. G. Blue can be promoted.

In an object oriented system, a high level routine requests an object to perform one of its methods by sending the object a "message" telling the object what to do. The receiving object responds to the message by choosing the method that implements the message name, executing this method and then returning control to the calling high level routine, along with the results of the method.

Object oriented programming systems may be employed as database management systems which are capable of operating upon a large database, and which are expendable and adaptable. In an object oriented database management system, the data in the database is organized and encapsulated in terms of objects, with the in-

stances of the objects being the data in the database. Similarly, the database manager may be organized as a set of objects with database management operations being performed by sending messages from one object to another. The target object performs the requested action on its attributes using its methods.

As described above, object oriented database management systems typically operate on large databases. However, it is difficult to manipulate the large database, or a large subset of the database which results from a database query, in order to view, update or delete selected elements therefrom. From a system perspective, the object oriented database, or the large query results, are a "boundless" data stream which is too large to fit in the system's memory, or into the portion of the system's memory allocated to an individual user.

As is well known to those having skill in the art, a data processor typically includes internal, volatile memory, often referred to as random access memory (RAM) or simply as "memory", which is available to the system for data manipulation. For multiuser systems, memory is typically divided among the users. Due to physical memory limitations, each user is limited to a maximum size of data stream which can be manipulated. In order to allow manipulation of data streams which exceed the maximum size, more memory must be provided, or a mechanism must be provided which creates the appearance of manipulating a boundless data stream without exceeding the physical limitations of the data processing environment.

Attempts have been made in prior art functionally programmed database management systems to provide the appearance of access to a boundless data stream by providing a "cursor". For example, in the Structured Query Language (SQL) database management system marketed by IBM Corpora tion as program product number 5470-XYR, a "cursor" is provided. The cursor is a file which provides forward pointers to a larger data stream. These forward pointers allow the data to be manipulated in the forward direction as one very large data stream. Operation of a cursor in an SQL database system is described in a publication entitled "IBM Database 2 Version 2 SQL Reference Release 1", IBM publication number SC26-4380-0, the disclosure of which is incorporated herein by reference.

Unfortunately an SQL cursor only allows a boundless data stream to be accessed or "scrolled" in the forward direction. Data manipulation often requires backward scrolling as well. In other words, bidirectional scrolling is required. Moreover, one particular SQL cursor may only be employed by one user at one time. Unfortunately, large database management systems often require multiuser capability. Finally, the SQL cursor is implemented in a functionally programmed database management system. A process and apparatus for bidirectional, multiuser manipulation of a boundless data stream in an object oriented database management system has heretofore not been available.

It is therefore an object of the invention to provide a process and apparatus for manipulating a boundless data stream.

It is another object of the invention to provide a process and apparatus for manipulating a boundless data stream in an object oriented programming system.

It is still another object of the invention to provide a process and apparatus for bidirectionally scrolling a boundless data stream in an object oriented programming system.

It is yet another object of the present invention to provide a process and apparatus for allowing a single user to bidirectionally scroll multiple instances of the same boundless data stream in an object oriented programming system.

These and other objects are provided by the present invention in an object oriented database management system including a data storage device having a database of data objects stored thereon and an object oriented database manager operating in a data processor connected to the data storage device. According to the invention, the object oriented database manager includes a "stream" class of objects. Each time a boundless data stream is manipulated by a user, an instance of the stream class is created.

The stream class of the present invention includes attributes and methods for allowing a boundless data stream to be manipulated bidirectionally by one user without exceeding the physical limitations of the user's environment. In particular, the stream class attributes include a number of pointers which identify sequential data objects selected from the database, and an attribute which identifies the maximum number of pointers permitted. The maximum number of pointers is limited by the amount of memory available to the user of the stream class instance. In effect, the attributes of the stream class create a "window" into the database for the user, with the maximum size of the window being determined by the amount of memory available to the user.

It will be understood by those having skill in the art that the pointer in the stream attributes may point directly to (i.e. directly identify) data objects in the database. Alternatively, the pointers may point to (identify) data objects in a large data stream produced as a result of a query to a database. In fact, the pointers may point to data elements in an SQL cursor. Each pointer in the stream class attribute may also indirectly point to

the data elements in the database by pointing to a stream element which includes therein one or more data elements from the database. Alternatively, each pointer in the stream class attribute may point to a stream element which in turn contains a pointer to one or more data elements from the database. In other words, direct or indirect pointing may be employed.

According to the invention, an instance of the stream class is created when a user desires to manipulate a boundless data stream. The instance will contain space for a number of pointers, the maximum number of which is determined by the amount of available memory.

The data attributes of the stream class of the present invention also include an identification of a current one of the pointers with the current pointer changing in response to a user request. Thus, each instance of the stream class will include a current value identification of one of the pointers in the set of pointers in the stream class instance.

The stream class of the present invention also includes methods associated therewith. In particular, "move to next" and "move to previous" methods are included. The "move to next" method changes the pointers in the stream class instance attributes to include the data element immediately following the current data element, and to include neighboring data elements of the immediately following data element in the remaining pointers, so that pointers to sequential data elements are included in the stream instance. When the maximum number of pointers are already present in the stream class instance, at least one pointer must be deleted before the immediately following data element may be inserted. Similarly, the "move to previous" method changes the pointers in the stream class instance attributes to include the data element immediately preceding the current data element, and to include neighboring elements of the immediately preceding data element in the remaining pointers, so that pointers to sequential data elements are included in the stream instance. When the maximum number of pointers are already present in the stream class instance, at least one pointer must be deleted before the immediately preceding data element may be inserted. The stream instance thereby provides a "sliding window" into the large data stream, with the sliding window including a desired data element, and as many surrounding data elements as are possible, consistent with the user's memory allocation.

The stream class of the present invention also includes methods to "move to first" and "move to last". In these methods, the pointers in the stream instance attributes are changed to include the first and last stream elements, respectively, and following or remaining elements respectively, up to the maximum allowed names of pointers. The stream class may also include an attribute defining the maximum number of data elements to be read each time the database is physically accessed, to thereby provide input/output buffering. In a preferred embodiment, the maximum number of pointers in the stream class instance attribute is preferably an integer multiple of the maximum number of data elements which can be read by the system at one time. The number of elements in the window is thereby a function of the number of elements which may be physically accessed at one time.

The present invention will now be described more fully hereinafter with reference to the accompanying drawings :

Figure 1 illustrates a schematic representation of an object.

Figure 2 illustrates a schematic representation of an example of an object.

Figure 3 illustrates the inheritance property of objects.

Figure 4 illustrates a schematic block diagram of an object oriented computer system according to the present invention.

Figure 5 illustrates a schematic block diagram of an object oriented program according to the present invention.

Figure 6 illustrates a representation of the stream class of the present invention.

Figure 7 illustrates a representation of the stream class of the present invention, as it appears to a user.

Figure 8 illustrates a representation of the maximum size of a stream class of the present invention.

Figure 9 illustrates input/output buffering in a stream class of the present invention.

Figure 10 illustrates stream elements in a stream class of the present invention.

Figure 11 illustrates a current value pointer in a stream class of the present invention.

Figures 12 through 15 illustrate a stream class of the present invention at various stages during operation of a first example of the present invention.

Figures 16 through 29 illustrate a stream class of the present invention at various stages during operation of a second example of the present invention.

Figures 30 through 34 illustrate a stream class of the present invention at various stages during operation of a third example of the present invention.

Figures 35 through 40 illustrate a stream class of the present invention at various stages during operation of a fourth example of the present invention.

In an object oriented computer system, work is

accomplished by sending action request messages to an object which contains (encapsulates) data. The object will perform the requested action on the data according to its predefined methods. The requestor of the action need not know what the actual data looks like or how the object manipulates it.

An object's class defines the types and meanings of the data and the action requests (messages) that the object will honor. The individual objects containing data are called instances of the class. Classes generally relate to real-world things. For example, "Parts" may be a class. The data elements (slots) of a part might be a part number, a status and a part type. The instances of this class represent individual parts, each with its own part number, status, and type information. The programs performing the requested actions are called methods of the class.

Object classes can be defined to be subclasses of other classes. Subclasses inherit all the data characteristics and methods of the parent class. They can add additional data and methods, and they can override (redefine) any data elements or methods of the parent class. While most messages are sent to object instances, the message that requests that a new instance be created is sent to an object class. The class will cause a new instance to be created and will return an object identifier by which that object will be known.

The sender of an action request message need not know the exact class of the object to which it is sending the message. As long as the target object either defines a method to handle the message or has a parent class that defines such a method, then the message will be handled using the data in the object instance and the method in its class or its parent class. In fact, it need not be an immediate parent, but may be a parent's parent, etc. The sender of the method need only have the object ID of the receiving object. This property of object oriented systems is called "inheritance". The inheritance property is used in the present invention.

Referring now to Figure 4, a schematic block diagram of an object oriented computer system 10 is illustrated. The system 10 includes a data processor 11 which may be a mainframe computer, minicomputer or personal computer. For large databases having multiple users, a mainframe computer is typically employed. As is well known to those having skill in the art, the data processor 10 includes a volatile data storage device 13, typically random access memory (RAM) for providing a working store for active data and intermediate results. Data in RAM 13 is erased when power to the data processor 11 is removed or a new user session is begun. System 10 also includes a nonvolatile data storage device 14 for permanent storage of objects. Device 14 may be a direct access storage device (DASD-a disk file) a tape file, an erasable optical disk or other well known device. Nonvolatile data storage device 14 will also be referred to herein as a "database". Volatile data storage device 13 will also be referred to as "memory". A display terminal 15 including a cathode ray tube (CRT) or other display, and a keyboard, is also shown.

An object oriented operating program 12 is also included in data processor 11. Object oriented operating program 12 may be programmed in object oriented languages such as "C" or "Smalltalk" or variations thereof, or in conventional programming languages such as FORTRAN or COBOL. The design of an object oriented operating program 12 is well known to those skilled in the art of object oriented programming systems, and will only be described generally below.

Referring now to Figure 5, the main components of an object oriented program (12, Figure 4) will be described. A more detailed description of the design and operation of an object oriented program is provided in "Object Oriented Software Construction", by Bertrand Meyer, published by Prentice Hall in 1988, the disclosure of which is incorporated herein by reference.

Referring to Figure 5, object oriented program 12 includes three primary components: a Messenger 51, an Object Management Table 52 and a Loaded Classes Table 53. The Messenger 51 controls communication between calling and called messages, Object Management Table 52 and Loaded Classes Table 53. Object Management Table 52 contains a list of pointers to all active object instances. The Loaded Classes Table 53 contains a list of pointers to all methods of active object classes.

Operation of the Object Oriented Program 12 will now be described for the example illustrated in Figure 5, in which Method A (block 54) of an object sends a message to Method B (block 55) of an object. Method A sends a message to Method B by calling Messenger 51. Messenger 51 obtains a pointer to the data frame 56 of the instance object specified by Method A, by searching Object Management Table 52 for the instance object. If the specified instance object cannot be found, Object Management Table 52 adds the instance object to the table and calls the instance to materialize its data from the database. Once in the instance table, Object Management Table 52 returns the pointer to the materialized instance object.

Messenger 51 then obtains the address of Method B from the Loaded Classes Table 53. If the instance's class is not loaded, the Loaded Classes Table 53 will load it at this time to materialize its data. The Loaded Classes Table 53 searches for

the specified method (Method B) and returns the address of the method to Messenger 51.

The Messenger 51 then calls Method B, passing it a system data area and the parameters from the call made by Method A including the pointer. Method B accesses the data frame 56 using the pointer. Method B then returns control to the Messenger 51 which returns control to Method A.

Inherent within object oriented database management systems is the necessity to easily manipulate very large amounts of data. Due to the physical limitations of the computer and its operating system, each user is limited to a finite amount of storage. This, in turn, limits the amount of data which can be in memory at any one moment. In order to increase the volume of accessible data, either more memory must be allocated to the application or a mechanism is needed which provides the appearance of access to a very large volume of data while still remaining within the physical limitations of the user's environment. The present invention, a process and apparatus of manipulating boundless data streams, provides the user with the ability to manipulate any number of lists of virtually unlimited size in a finite amount of storage. This invention expands on the "cursor" known to those skilled in the art which permits only uni-directional access to data and which is non-recursive in that the data can only be used once by a single user at any single point in time, e.g. the SQL cursor.

The boundless data steam expands upon the capabilities of a "cursor" by providing more flexible navigation, i.e. bidirectional movement. Memory and I/O optimization in the boundless data stream can be customized to meet the requirements of a particular application in various ways including altering the size of the stream as well as the amount of data read into memory from the cursor at one time. Furthermore, the data stream transparently manages multiple data streams against the same "cursor". Thus, the boundless data stream reduces the amount of computer application code necessary to manipulate a "cursor" while providing a superior interface to the data.

In an object oriented programming system, according to the present invention, the boundless data stream is an infinitely long linked list of object references. The physical structure of the data stream in memory may be represented as illustrated in Figure 6. The large, outer stream box 21 is the maximum amount of memory utilized by the stream. The smaller, inner stream boxes 22 are the number of rows read from the cursor 23 at any one time. The arrows show the direction in which one may access the data. The stream, however, logically appears to the user as a bi-directional cursor. This is illustrated in Figure 7 where the arrows show the direction in which one may access the data. The rows appear in groups of two because, in this example, the data will be read into the stream two elements at a time. This number is arbitrary and can be designated by the user in order to optimize I/O.

In an object oriented programming system, the boundless data stream may be implemented by two object classes, namely the Stream class and the Stream Element class. The Stream Element class is where the actual data is stored about the individual data objects for which the stream was developed. It could be something as simple as an object ID to a join of any number of objects. These appear in memory only while the Stream class needs them. They are discarded and recreated as memory requirements dictate.

The Stream class is composed of a linked list of object references to Stream Elements. Additionally, it has all those methods necessary for converting a "cursor" or any other similar uni-directional database access mechanism into a boundless linked list. This linked list allows both forward and backward movement and transparently handles all the cursor interfacing. Essentially, a user just moves up and down the Stream and accesses the data directly. The Stream handles all the cursor opens, closes, and positioning as well as all the memory management chores. Thus, for all practical purposes, the Stream appears to the user as a "smart" linked list. That is, all the data within the Stream appears automatically. The user need only reference it.

The Stream class allows a user to create any number of lists based upon the same cursor. The user can move to the first, last, next, or previous elements. In addition, the user can specify the maximum number of Stream Elements to maintain in memory at any one time. He also can specify the number of elements to be read with each physical access of the database, i.e. designate I/O buffer size. Finally, the user can refresh the list at any time with a single instruction so that the list contains the current, updated value of the data as stored in the database.

Some of the attributes and methods of the stream class and stream element class of the present invention will now be described in detail.

ATTRIBUTES

**Max_List_Size**

Max_List_Size, an instance attribute (sometimes referred to as maximum attribute), permits the user to specify the maximum number of elements to keep in the linked list in RAM at any

one time. Through its use, the overall memory requirements of a stream are reduced. For example, if the number of elements in the list is equal to Max_List_Size and an additional element is to be added to the end of the list, then a Delete_First is done by the Move_to_Next method. The default value for Max_List_Size is LONG_MAX. Where Max_List_Size is reduced to less than this default, it is desirable to set the new value to a multiple of the Number_to_Read instance attribute. It is preferable to use an integer multiple greater than 1 such as 10 or 100 times Number_to_Read.

**Number_to_Read**

Number_to_Read attribute (sometimes referred to as number attribute) is the number of rows to be read from the cursor at any one time, i.e. the small box 22 in Figure 6. This may also be called Buffer_Size. By reading a set number of rows at any given time, physical I/O performance is optimized.

**Nb_Elements**

The Nb_Elements instance attribute contains the total number of elements in the logical linked list. It has three states:

**-- 'zero'**

Nb_Elements is set to "zero" when the stream has just been created but the database has not been accessed.

**-- 'Unknown_Nb_Elem' (999999999)**

This is known as the "unknown" state since although elements have been read from the database, the actual end of the logical linked list has not been reached, i.e. the end of the cursor has not been reached.

**-- 'Known'**

This state is the actual number of elements in the logical linked list. This state comes in to existence only when the end of the cursor has been reached, and only then can it be assigned to Nb_Elements.

An alternative embodiment adds an additional select statement which actually returns the count of the number of elements which will meet the selection criteria and assign that value at that time.

**Current_Cursor**

The Current_Cursor instance attribute is used to indicate which cursor is the active cursor for the stream instance. Although it is possible to define any number of cursors within a particular stream class, any stream instance can only use one cursor within a particular stream class at a time.

**Current_Value_Pointer**

Current_Value_Pointer (sometimes referred to as current attribute) is the OREF to the Current Stream Element in the linked list.

**Row_Count**

Row_Count refers to the count of the rows in the database or in the result of the selection criteria.

METHODS

**Current_Value**

The Current_Value instance method, returns the OREF to the current Stream Element in the linked list.

**Move_to_First**

The Move_to_First instance method positions the user at the first element in the logical linked list.

**Move_to_Last**

The Move_to_Last instance method positions the user at the last element in the logical linked list.

**Move_to_Next**

The Move_to_Next instance method positions the user at the next element in the logical linked list.

**Move_to_Prev**

The Move_to_Prev instance method positions the user at the previous element in the logical linked list.

**Restart**

The Restart instance method closes the physical database access link, i.e. the "cursor". It deletes all stream elements from memory and clears out the physical linked list. Finally, it resets the stream instance attributes to their original state. In other words, it forces the stream to refresh itself without having to delete it and create a new stream instance.

**Reopen**

The reopen method reopens a closed cursor and sets the pointer to the beginning of the cursor.

Other methods and attributes which are utilized to implement the above methods will be described in the examples which follow.

STREAM ELEMENT CLASS--ATTRIBUTES

**Object_Id**

The Object_Id instance attribute is an OREF to any object.

STREAM ELEMENT CLASS--METHODS

**Create**

This method creates a new Stream Element object.

**Initialize**

This method stores the values in the Stream Element object's instance attributes.

Examples are now presented to illustrate the operation of the invention. Example 1 describes the move to first method which manipulates the data stream to set the current value pointer to point at the first element in the cursor. Example 2 describes the move to last method which manipulates the data stream to set the current value pointer to point at the last element in the cursor. Example 3 describes the move to next method which manipulates the data stream to set the current value pointer to point at the next element in the cursor. Example 4 describes the move to previous method which manipulates the data stream to set the current value pointer to point at the previous element in the cursor.

For purposes of these examples, arbitrary values are chosen for various attributes including Max_List_Size = 12, Number_to_Read = 4, and 19 rows in the database meet the selection criteria. In all examples, the term "database" is used to refer to nonvolatile data storage device 14 (Figure 4). The terms "RAM" and "memory" are used to refer to a volatile data storage device 13 (Figure 4).

The term "row" refers to an entry in a database table. It can represent either the physical row in the database table or the row as stored in memory. Thus, it is sometimes used interchangeably with the term Stream Element. "Stream Element" refers to an object instance which is pointed to by an object reference (OREF) within the physical list. OREF's are commonly used in object oriented programming systems to point to an instance of an object. Stream Elements are the memory images of rows retrieved from the database.

"Logical list", "logical linked list", or "virtual list" is the entire list as defined by the cursor's selection criteria against the physical database table(s). This is the list which "appears" to the user to be in memory but in reality is not. If instance attribute Max_List_Size, were set to infinity and all rows from the cursor could be placed into memory at one time, then the logical list would be the same as the physical list. "Physical list", "physical linked list", or "actual list" refer to the actual in-memory linked list used to track the Stream Elements. This is the "sliding window" of the database. Its size is limited by the Max_List_Size instance attribute. That is, the number of elements in this linked list is bounded by: 1 <= number of elements in the physical list <= Max_List_Size.

"Maximum list size" or "max-size" refer to the value defined by the Max_List_Size instance attribute. This is the maximum number of rows in memory at any one time. If the number of rows in the physical list exceeds this number, the first row in the physical list must be removed from memory before an additional row may be added at the end of the physical list. "Number to read", "read buffer", "burst mode read", or "burst read" refer to the value defined by the Number_to_Read instance attribute. A certain number of rows are read each time the stream must access the database with I/O commands. Since most database access mecha-

nisms provide for some sort of data buffering, selection of the buffer size for this burst read optimizes the utilization of the database's buffering.

The reserved word SELF appears throughout the examples. As is well known to those having skill in the art, the use of SELF in an expression having a method and object denotes the current instance of the class. The function of SELF in the present invention denotes the current instance of the Stream Class.

Deferred methods also appear throughout the examples. As is well known to those having skill in the art, a deferred method which is defined by a class but is only implemented by the dependents (children) of the class. The function of the deferred method is precisely specified by the Stream Class of the present invention, although it is implemented by children of the Stream Class.

Memory utilization at a particular time ("flash points") are also conceptually represented in the Examples. The data stream is illustrated as a vertical box as shown, for example in Figure 6 and 7. The memory occupied by the physical list is illustrated as a long horizontal box as depicted in Figure 8. It will hold at most Max_List_Size elements. The memory occupied by the elements read in any one burst mode from the database will be shown as a short horizontal box, usually within the physical list box, as illustrated in Figure 9. Each burst mode box will hold a maximum of Number_to_Read elements.

As illustrated in Figure 10, row numbers will often be seen within the burst read boxes to demonstrate which rows, i.e. stream elements, are in memory at that time. Finally, the current value pointer will be represented by a vertical arrowhead underneath the physical list. As illustrated in Figure 11, it will point to the current element. In this figure, it is pointing to element 05.

## EXAMPLE 1--MOVE TO FIRST

As a first example, assume that there is a stream My_Stream which has already been created but has not yet been accessed. The physical list in memory is illustrated in Figure 12. This example will load the first Number_to_Read elements which the buffer can hold. Once these are in the physical list, the current value pointer is set to point to the first element.

A user of My_Stream issues a call to the Move_to_First method, and the Move_to_First method receives control. This call can be represented as:

My_Stream.Move_to_First

The Move_to_First method determines that this is the first time the stream has been accessed. Thus, it makes a call to the stream's Load_Next method. Load_Next invokes the Fetch_Row method to retrieve a stream element. Fetch_Row now receives control. It determines that this is the first time this stream has been accessed and issues a call to the Open method. Open receives control and invokes the deferred method for the appropriate database query language to open the stream. This deferred method receives control.

The operation to this point can be illustrated on the program invocation stack as:

My_Stream.Move_to_First, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.Open, which in turn called
SELF.database_query_language_Open

The deferred method loads the relevant search criteria and opens the actual data cursor, i.e. cursor 23 in Figure 6. It then returns a result code to its caller. Assuming that the open by the deferred method is successful, OK is returned to the caller, i.e. the Open method. Open determines that the open was successful and initializes instance attributes. Cursor_Status is set to 'open' and Nb_Elements is set to Unknown_Nb_Elems. The Row_Count instance attribute is set to 0. Open then returns to its caller, i.e. Fetch_Row. Fetch_Row now issues a call to the database_query_language to Fetch, i.e. to get a stream element. Fetch receives control. The program invocation stack can be illustrated as:

My_Stream.Move_to_First, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database-query_language_Open

The deferred Fetch routine fetches cursor row 01 into the appropriate host variables and issues a call to the Stream Element's Create method. The Stream Element's Create method creates an instance of the Stream Element and invokes the Initialize method. The program invocation stack can be illustrated as:

My_Stream.Move_to_First, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch, which in turn called
Stream_Element.Create, which in turn called
Stream_Element.Initialize.

The Stream Element's Initialize method stores the values in its instance attributes and returns to the Create method. The Stream Element's Create method returns control to its caller, database_query_language_Fetch. Fetch returns the OREF of the newly created Stream Element to its caller, Fetch_Row. The Fetch_Row method increments the Row_Count instance attribute from 0 to 1 and returns control to its caller, Load_Next.

The Load_Next method adds the stream element to the end of the list which is currently empty. The physical list is illustrated in Figure 13. The Load_Next method then invokes Fetch_Row and adds the resultant stream element to the physical list three more times, for a total of Number_to_Read times, in this example, 4 times. The physical list can be represented in Figure 14.

Load_Next has caused Number_to_Read, i.e. 4, rows to be read from the database. It now returns control to its caller, Move_to_First. At this point, the Current_Value_Pointer is undefined. Thus, the Move_to_First method now establishes the Current_Value_Pointer to point to the first element in the list. The physical list is illustrated in Figure 15. Note that 4 elements were read in at one time in a single burst. These are illustrated in Figure 15 in the burst read box (read buffer). The Current_Value_Pointer is pointing to the first element, stream element 01.

EXAMPLE 2--MOVE TO LAST

For the second example, again assume that there is a stream My_Stream which has already been created but has not yet been accessed. The physical list is illustrated in Figure 16. This example reads in the elements a buffer at a time. Since the size of memory is 12 and buffer size is 4, three buffers will be read. Another buffer is read. Memory is then full. The first element in memory will be deleted and the next element from the buffer will be loaded at the end of memory. This will be repeated until the buffer is empty. Another buffer will be read from the database. Elements from the front of the physical list will be deleted and Elements from the buffer will be added to the end of the physical list. This process will continue until the end of the cursor is reached. The Current_Value_Pointer will then be set to point to the last element in the physical list.

A user of My_Stream issues a call to the Move_to_Last method and the Move_to_Last method receives control. The call can be represented as:

**My_Stream.Move_to_Last**

Move_to_Last determines that this is the first time the stream has been accessed. Thus, it calls the stream's Load_Next method. Load_Next invokes the Fetch_Row method to retrieve a stream element. Fetch_Row receives control. It determines that this is the first time this stream has been accessed and calls the Open method. Open receives control and invokes the deferred database_query_language_Open method. This deferred Open method receives control. The program invocation stack can be represented as:

My_Stream.Move_to_Last, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.Open, which in turn called
SELF.database_query_language_Open

Deferred Open loads the relevant search criteria and opens the actual data cursor, i.e. curser 23 in Figure 6. It then returns a result code to its caller, Open. Assuming that the open is successful, OK is returned to the caller, Open. Open determines that the open was successful and initializes the instance attributes. Cursor_Status is set to 'open' and Nb_Elements is set to Unknown_Nb_Elems. The Row_Count instance attribute is set to Open then returns control to its caller, Fetch_Row. Fetch_Row calls database_query_language_Fetch, to get a stream element. Fetch receives control. The program invocation stack can be represented as:

My_Stream.Move_to_Last, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch

Fetch fetches cursor row 01 into the appropriate host variables and issues a call to the Stream_Element's Create method. The Stream_Element's Create method creates an instance of the stream element and invokes the Initialize method. The program invocation stack can be represented as:

My-Stream.Move_to_Last, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch, which in turn called
Stream_Element.Create, which in turn called
Stream_Element.Initialize.

The Stream Element's Initialize method stores the values in its instance attributes and returns control to the Create method. The Stream Element's Create method returns control to its caller database_query_language_Fetch. Fetch returns the OREF of the newly created Stream Element to its caller, Fetch_Row. Fetch_Row increments the Row_Count instance attribute from 0 to 1 and returns control to its caller_Load_Next.

The Load_Next method adds Stream Element 01, i.e. Stream Element from Row 01 of the database selected results, to the end of the list which is currently empty. The physical list is illustrated in Figure 17. Load_Next then invokes Fetch_Row and adds the resultant stream element to the physical list three more times, for a total of Number_to_Read times, i.e. 4 times. Thus, 4 elements are read in a single burst. The physical list can is illustrated in Figure 18.

Load_Next has caused Number_to_Read rows, i.e. 4 rows, to be read from the database. It

now returns control to its caller, Move_to_Last. Move_to_Last determines that the end of the cursor has not been reached. Thus, it calls the Load_Next method again, to load the next Number_to_Read block of rows. Load_Next invokes the Fetch_Row method to retrieve a stream element. Fetch_Row calls database_query_language_Fetch, to get a stream element. Fetch receives control. The program invocation stack can be represented as:

My_Stream.Move_to_Last, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch.

Fetch fetches cursor row 05 into the appropriate host variables and issues a call to the Stream Element's Create method. The Stream Element's Create method creates an instance of the Stream Element and invokes the Initialize method. The program invocation stack can now be represented as:

My_Stream.Move_to Last, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch, which in turn called
Stream_Element.Create, which in turn called
Stream_Element.Initialize.

The Stream Element's Initialize method stores the values in its instance attributes and returns control to its caller, the Create method. The Stream Element's Create method returns control to its caller, database_query_language_Fetch. The Fetch routine returns the OREF of the newly created Stream Element to its caller, Fetch_Row.

Fetch_Row increments the Row_Count instance attribute from 4 to 5 and returns control to its caller, Load_Next. Load_Next adds the 05 Stream_Element, i.e. the element from row 5 of the cursor, to the end of the list. The physical list is illustrated in Figure 19. Load_Next then invokes Fetch_Row and adds the resultant stream element to the physical list three more times, for a total of Number_to_Read times, i.e. 4 times. This is because 4 elements are read in a single burst. The physical list is illustrated in Figure 20. Load_Next has caused Number_to_Read rows, i.e. 4 rows, to be read from the database. It now returns control to its caller, Move_to_Last.

Move_to_Last determines that the end of the cursor has not been reached. Thus, it calls the Load_Next method again, to load the next Number_to_Read block of rows, i.e. 4 rows. The Load_Next invokes the Fetch_Row method to retrieve a stream element. Fetch_Row issues a call to database query language Fetch. Fetch receives control. The program invocation stack can be represented as:

My-Stream.Move_to_Last, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch.

Fetch fetches cursor row 09 into the appropriate host variables and issues a call to the Stream Element's Create method. The Stream Element's Create method creates an instance of the Stream Element and invokes the Initialize method. The program invocation stack can be represented as:

My_Stream.Move_to_Last, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch, which in turn called
Stream_Element.Create, which in turn called
Stream_Element.Initialize.

The Stream Element's Initialize method stores the values in its instance attributes and returns control to the Create method. The Stream Element's Create method returns control to its caller, database_query_language_Fetch. Fetch returns the OREF of the newly created Stream Element to its caller, Fetch_Row.

Fetch_Row increments the Row_Count instance attribute from 8 to 9 and returns control to its caller, Load_Next. Load_Next adds the 09 Stream Element to the end of the list. The physical list is illustrated in Figure 21.

Load_next then invokes Fetch_Row and adds the resultant Stream_Element to the physical list three more times, for a total of Number_to_Read times, i.e. 4 times. This is because 4 rows are read in a single burst. The physical list is illustrated in Figure 22. Load_Next has caused Number_to_Read rows, i.e. 4 rows, to be read from the database. It now returns control to its caller, Move_to_Last.

Move_to_Last determines that the end of the cursor has not been reached, so it issues a call to the Load_Next method again, to load the next Number_to_Read block of rows, i.e, 4 rows. Load_Next invokes Fetch_Row to retrieve a stream element. Fetch_Row calls database_query_language_Fetch, to get a Stream Element. Fetch receives control. The program invocation can be represented as:

My_Stream.Move_to_Last, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch.

Fetch fetches cursor row 13 into the appropriate host variables and issues a call to the Stream Element's Create method. The Stream Element's Create method creates an instance of the Stream Element and invokes the Initialize method. The program invocation stack can be represented

as:
My-Stream.Move_to Last, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch, which in turn called
Stream_Element.Create, which in turn called
Stream_Element.Initialize.

The Stream Element's Initialize method stores the values in its instance attributes and returns control to the Create method. The Stream Element's Create method returns control to its caller, Fetch. The Fetch routine returns the OREF of the newly created Stream Element to its caller, Fetch_Row.

The Fetch_Row method increments the Row Count instance attribute from 12 to 13 and returns control to its caller, Load Next. Load_Next determines that the physical list has reached the Max_List_Size boundary, so it deletes the first element off the list. The physical list is illustrated in Figure 23. Load_Next next adds the stream element to the end of the list. The physical list is illustrated in Figure 24. Notice that no more than a total of Max_List_Size, i.e. 12, elements are in the list at any one time.

Load_Next invokes Fetch_Row and drops off the front elements and adds the resultant Stream Elements to the physical list three more times, for a total of Number_to_Read times. As is well known to those having skill in the art, elements are deleted from the front of a list by freeing the memory. The physical list is illustrated in Figure 25. Load_Next has caused Number_to_Read rows, i.e. 4 rows, to be read from the database. It now returns control to its caller, Move_to_Last.

Move_to_Last determines that the end of the cursor still has not been reached. Thus, it calls the Load_Next method again, to load the next Number_to_Read block of rows, i.e. 4 rows. Load_Next invokes Fetch_Row to retrieve a Stream Element. Fetch_Row calls database_query_language_Fetch, to get a Stream Element. Fetch receives control. The program invocation stack can be represented as:
My-Stream.Move_to_Last, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch.

Fetch fetches cursor row 17 into the appropriate host variables and issues a call to the Stream Element's Create method. The Stream Element's Create method creates an instance of the Stream Element and invokes the Initialize method. The program invocation stack can be represented as:
My_Stream.Move_to_Last, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch, which in turn called
Stream_Element.Create, which in turn called
Stream_Element.Initialize.

The Stream Element's Initialize method stores the values in its instance attributes and returns control to the Create method. The Stream Element's Create method returns control to its caller, Fetch. Fetch returns the OREF of the newly created Stream Element to its caller, Fetch_Row. Fetch_Row increments the Row Count instance attribute from 16 to 17 and returns to its caller, Load_Next. Load_Next determines that the physical list has reached the Max_List_Size boundary, i.e. 12, so it deletes the first element off the list. The physical list is illustrated in Figure 26. Load_Next adds the stream element to the end of the list. The physical list is illustrated in Figure 27. Notice that no more than a total of Max_List_Size, i.e. 12, elements are in the list at any one time. Load next invokes Fetch_Row and drops off the front elements and adds the resultant Stream Element to the physical list two more times. The physical list is illustrated in Figure 28.

The Load_Next method invokes the Fetch_Row method a fourth time. Load_Next invokes the Fetch_Row method to retrieve a Stream Element. Fetch_Row calls database_query_language_Fetch, to get a Stream Element. Fetch receives control. The program invocation stack can be represented as:
My_Stream.Move_to_Last, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch.

Fetch receives an End_of_Cursor from the fetch and returns control to Fetch_Row with the appropriate indicator. The Fetch_Row method sets Nb_Elements equal to Row_Count, sets the Cursor Status attribute to End of Cursor and returns control to its caller, Load_Next. The Load_Next method determines that the cursor has been closed from the value in Cursor_Status and returns control to its caller, Move_to_Last, without modifying the physical list. The Move_to_Last method determines that the end of the cursor has now been reached. Since the Current_Value_Pointer is currently undefined, Move_to_Last establishes the Cur rent_Value_Pointer to point to the last element in the list. The physical list is illustrated in Figure 29. Note that there are only Max_Size_List elements, i.e. 12 elements, in the list. Also note that the Current_Value_Pointer is pointing to the last element, i.e. the element from row 19 in the result of the database selection.

EXAMPLE 3--MOVE TO NEXT

For purposes of this example, assume that a partial list is currently in memory. The physical list is illustrated in Figure 30 where the maximum 12 elements (Max_List_Size) are in memory and the Current_Value_Pointer is pointing to the last element in the physical list, i.e. the element from the row 16 in the database. In this example, since the Current_Value_Pointer is pointing to the last element in the physical list, the next Number_to_Read elements will be read from the cursor into the buffer. Elements will be deleted from the front of the physical list and added to the end of the physical list from the buffer, one at a time until the contents of the entire buffer is in the physical list. The Current_Value_Pointer will then be set to point to the next element in the physical list.

A user of My_Stream issues a call to the Move_to_Next method and the Move_to_Next method receives control. The call can be represented as:

My_Stream.Move_to_Next ( End_of_List_Indicator )

End_of_List_Indicator is passed as a parameter and is assigned a state as control is returned. Move_to_Next determines that the Current_Value_Pointer is on the last element, i.e. the 12th element, in the physical list and also that the End_of_the_Cursor has not been reached. Thus, it invokes Load_Next to load in the next buffer, i.e. 4 elements, from a single burst. Load_Next invokes the Fetch_Row method to retrieve a Stream Element. Fetch_Row calls database_query_language_Fetch, to get a Stream Element. Fetch receives control. The program invocation stack can be represented as:

My_Stream.Move_to_Next, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database query language Fetch.

Fetch fetches cursor row 17 into the appropriate host variables and issues a call to the Stream Element's Create method. The Stream Element's Create method creates an instance of the Stream Element and invokes the Initialize method. The program invocation stack can be represented as:

My-Stream.Move_to_Next, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch, which in turn called
Stream_Element.Create, which in turn called
Stream_Element.Initialize.

The Stream Element's Initialize method stores the values in its instance attributes and returns control to the Create method. The Stream Element's Create method returns control to its caller, Fetch. Fetch returns the OREF of the newly created Stream Element to its caller, Fetch_Row.

The Fetch_Row method increments the Row_Count instance attribute from 16 to 17 and returns control to its caller, Load_next. The Load_Next method determines that the physical list has reached the Max_List_Size boundary, i.e. 12 elements. Thus, it deletes the first element off the list. The physical list is illustrated in Figure 31.

The Load_Next method adds the Stream Element to the end of the list. The physical list is illustrated in Figure 32. Notice that no more than a total of Max_List_Size, i.e. 12, elements are in the list at any one time. The Load_Next method then invokes Fetch_Row and drops off the front elements and adds the resultant Stream Element to the physical list two more times. The physical list is illustrated in Figure 33.

The Load_Next method invokes the Fetch_Row method a fourth time. The Load_Next invokes the Fetch_Row method to retrieve a Stream_Element. Fetch_Row calls database_query_language_Fetch, to get a Stream Element. Fetch receives control. The program invocation stack can be represented as:

My-Stream.Move_to_Next, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch.

Fetch receives an End_of_Cursor from the fetch and returns control to Fetch_Row with the appropriate End_of_Cursor indicator. The Fetch_Row method sets the Cursor_Status attribute to End_of_Cursor and returns control to its caller, Load_Next. The Load_Next method determines that the cursor has been closed from the value in Cursor_Status and returns control to its caller, Move_to_Next, without modifying the physical list. The Move_to_Next method then moves the Current_Value_Pointer to the next element in the list, i.e. the element that was row 17 in the database. The physical list is illustrated in Figure 34. Notice that there are Max_Size_List elements, i.e. 12 elements, in the physical list and the current value pointer is pointing to the element which was in row 17 in the database.

EXAMPLE 4--MOVE TO PREV

For purposes of this example, assume that a partial list is currently in memory with the Current_Value_Pointer pointing to the first element of the list. The physical list is illustrated in Figure 35. In this example, it will be determined that Current_Value_Pointer is set to the first ele-

ment of the physical list. The cursor pointer will be determined. All elements in the physical list will be deleted. The cursor will then be closed. The cursor will be reopened with the pointer pointing to the beginning of the cursor. Memory will be loaded with elements a Number_to_Read elements at a time. The Current_Value_Pointer will then be set to point to the previous element in memory (the physical list).

A user of My_Stream issues a call to the Move_to_Prev method and the Move_to_Prev method receives control. The call can be represented as:

My_Stream.Move_to_Prev ( End_of_List_Indicator )

End_of_List_Indicator is passed as a parameter and is assigned its state as control is returned. The Move_to_Prev method determines that the Current_Value_Pointer is on the first element in the physical list and that some elements have previously been dropped from the beginning of the list. This is determined by comparing Row Count to Max_List_Size. If it is greater, then some elements have been deleted from the beginning of the list. Thus, it invokes Load_Prev to load in the previous buffer. The Load_Prev method determines the physical row number of the element as it resided in the result of the database selection immediately preceding the first element in the physical list, i.e. row number 4. It then closes the cursor by calling the Close method.

Close calls the database_query_language_Close method. If there is more than one cursor open, database_query_language_Close determines which is open, closes it, and returns control to the Close method. Close returns control to its caller, Load_Prev. The Load_Prev method then deletes all the physical list elements and frees memory. The physical list is illustrated in Figure 36.

The Load_Prev method then executes the Load_Next method for Max_List_Size / Number_to_Read times since Max_List_Size is 12 and Number_to_Read is 4, Load_Prev executes Load_Next 3 times. For the first of the 3 times, Load_Next receives control on the first loop and invokes the Fetch_Row method to retrieve a Stream Element. Fetch_Row calls database_query_language_Fetch, to get a Stream Element. Fetch receives control. The program invocation stack can be represented as:

My Stream.Move to_Prev, which in turn called
SELF.Load_Prev, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch.

Fetch finds that the cursor is closed and returns to its caller with an appropriate closed cursor indication. The Fetch_Row method, upon finding that the cursor is closed, calls the Reopen method to reopen and reposition the cursor. The Reopen method opens the cursor and positions the pointer to the beginning of the cursor. It then returns to the Fetch_Row method.

Fetch_Row now issues a call to itself to call database_query_language_Fetch, to get a Stream Element. Fetch receives control. The program invocation stack can be represented as:

My_Stream.Move_to_Prev, which in turn called
SELF.Load Prev, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch.

Fetch fetches cursor row 01 into the appropriate host variables and issues a call to the Stream Element's Create method. The Stream Element's Create method creates an instance of the Stream Element and invokes the Initialize method. The program invocation stack can be represented as:

My_Stream.Move-to_Prev, which in turn called
SELF.Load_Prev, which in turn called
SELF.Load_Next, which in turn called
SELF.Fetch_Row, which in turn called
SELF.Fetch_Row, which in turn called
SELF.database_query_language_Fetch, which in turn called
Stream_Element.Create, which in turn called
Stream_Element.Initialize.

The Stream Element's Initialize routine stores the values in its instance attributes and returns to the Create method.

The Stream Element's Create method returns control to its caller, database_query_language_Fetch. Fetch returns the OREF of the newly created Stream Element to its caller, the second Fetch_Row. The second Fetch_Row method increments the Row_Count instance attribute from 0 to 1 and returns control to its caller, the first Fetch_Row. The first Fetch_Row returns control to Load_Next.

The Load_Next method adds the Stream Element to the end of the list, which is currently empty. The physical list is illustrated in Figure 37. The Load_Next method then invokes Fetch_Row and adds the resultant Stream Element to the physical list three more times, for a total of Number to Read times. The physical list is illustrated in Figure 38. Load_Next has caused Number_to_Read rows, i.e. 4 rows, to be read from the database since Number_To_Read is 4. It returns control to its caller, Load_Prev. Load_Prev invokes Load_Next two more times,

for a total of three times. The physical list is illustrated in Figure 39. Load_Next is called 3 times because Max_List_Size is 12 and Number_to_Read is 4.

Load_Prev now returns control to its caller, Move_to_Prev. The Move_to_Prev method positions the Current_Value_Pointer to the appropriate element in the list. The physical list is illustrated in Figure 40. Notice that the Current_Value_Pointer is pointing to the element from the 4th row in the database, i.e. one before the row number from the database which the pointer pointed to in the physical list (row 5).

## PSEUDO CODE APPENDIX A - STREAM CLASS

The following Appendix contains a pseudo code listing of an implementation of the stream class of the present invention in an object oriented computer system. The pseudo code listing is designed to operate with IBM's well-known Structured Query Language (SQL).

### Claims

1. A process for manipulating a data stream comprising a second plurality of data objects in an object oriented database management system comprising a data storage device, a database of a first plurality of data objects stored in said data storage device in a predetermined sequence, a data processor connected to said data storage device, and an object oriented database manager operating in said data processor, for manipulating said first plurality of data objects; said process comprising the steps of:
providing a stream class of objects in said object oriented database manager, said stream class of objects including stream class attributes and stream class methods;
said stream class attributes comprising:
a plurality of pointers for identifying a second plurality of data objects selected from said first plurality of data objects in said database;
a current attribute for identifying a current one of said plurality of pointers; and
a maximum attribute for identifying the maximum number of pointers in said plurality of pointers;
said stream class methods comprising:
a first method for placing in an instance of said stream class, said plurality of pointers including a pointer for identifying a data object immediately preceding the data object identified in said current attribute;
a second method for placing in an instance of said stream class, said plurality of pointers including a pointer for identifying a data object immediately succeeding the data object identified in said current attribute; and
creating an instance of said stream class object to manipulate said first plurality of data objects using said plurality of pointers.

2. The process according to Claim 1 characterized in that said stream class methods further comprise:
a third method for placing in an instance of said stream class, said plurality of pointers including a pointer for identifying a first data object in the predetermined sequence of said first plurality of data objects; and
a fourth method for placing in an instance of said stream class, said plurality of pointers including a pointer for identifying a last data object in the predetermined sequence of said first plurality of data objects.

3. The process according to Claim 1 characterized in that said plurality of pointers directly identify said second plurality of data objects.

4. The process according to Claim 1 characterized in that said plurality of pointers comprise said second plurality of data objects.

5. The process according to Claim 1 further characterized in that it comprises the step of creating a stream element class of objects, and creating a second plurality of instances of said stream element class of objects, a respective one of said plurality of pointers identifying a respective one of said second plurality of instances of said stream element class.

6. The process according to Claim 5 characterized in that said second plurality of instances of said stream element class of objects includes a pointer for identifying a respective one of said second plurality of data objects.

7. The process according to Claim 5 characterized in that said second plurality of instances of said stream element class of objects comprises a respective one of said second plurality of data objects.

8. The process according to Claim 1 further characterized in that it comprises the step of selecting said maximum attribute to be less than or equal to the maximum number of pointers which may be stored in said data storage device.

9. The process of Claim 1 wherein said first method performs the following steps:
deleting at least one of said plurality of pointers from said stream class attributes; and
inserting said pointer for identifying a data object immediately preceding the data object identified in said current attribute, into said stream class attributes.

10. The process of Claim 1 wherein said second method performs the following steps:
deleting at least one said plurality of pointers from

said stream class attributes; and
inserting said pointer for identifying a data object immediately succeeding the data object identified in said current attribute, into said stream class attributes.

11. The process of Claim 2 wherein said third method performs the following steps:
deleting all of said plurality of pointers from said stream class attributes; and
inserting said pointer for identifying said first data object into said plurality of pointers.

12. The process of Claim 2 wherein said third method performs the following steps:
deleting all of said plurality of pointers from said stream class attributes; and
inserting said plurality of pointers for identifying said stream class, with the inserted plurality of pointers beginning with said first data object.

13. The process of Claim 2 wherein said fourth method performs the following steps:
deleting all of said plurality of pointers from said stream class attributes; and
inserting said plurality of pointers for identifying said stream class, with the inserted plurality of pointers ending with said last data object.

14. The process of Claim 1 wherein said step of creating an instance comprises the step of creating an in stance of said stream class object for each user of said object oriented database management system.

15. The process of Claim 1 wherein said step of creating an instance comprises the step of creating an in stance of said stream class object for each query of said database in said object oriented database management system.

16. An apparatus in an object oriented database management system comprising:
a data storage device;
a database of a first plurality of data objects stored in said data storage device in a predetermined sequence;
a data processor connected to said data storage device; and
an object oriented database manager operating in said data processor, for manipulating said first plurality of data objects, said object oriented database manager including a stream class of objects, said stream class of objects including stream class attributes and stream class methods;
said stream class attributes comprising:
a plurality of pointers for identifying a second plurality of data objects selected from said first plurality of data objects in said database;
a current attribute for identifying a current one of said plurality of pointers; and
a maximum attribute for identifying the maximum number of pointers in said plurality of pointers;
said stream class methods comprising:
a first method for placing in an instance of said stream class, said plurality of pointers including a pointer for identifying a data object immediately preceding the data object identified in said current attribute; and
a second method for placing in an instance of said stream class, said plurality of pointers including a pointer for identifying a data object immediately succeeding the data object identified in said current attribute;
said object oriented database manager further including means for creating an instance of said stream class object;
whereby said first plurality of data objects are manipulated using said plurality of pointers.

17. The apparatus according to Claim 16 characterized in that said stream class methods further comprise:
a third method for placing in an instance of said stream class, said plurality of pointers including a pointer for identifying a first data object in the predetermined sequence of said first plurality of data objects; and
a fourth method for placing in an instance of said stream class, said plurality of pointers including a pointer for identifying a last data object in the predetermined sequence of said first plurality of data objects.

18. The apparatus according to Claim 16 characterized in that said database of data objects comprises a subset of a larger database of data objects, said first plurality of data objects resulting from a query of said larger database.

19. The apparatus according to Claim 16 characterized in that said data storage device comprises a nonvolatile data storage device and wherein said database is stored in said nonvolatile data storage device.

20. The apparatus according to Claim 16 characterized in that said data storage device comprises a volatile data storage device and wherein said data base is stored in said volatile data storage device.

21. The apparatus according to Claim 16 characterized in that said subset of a larger database comprises a cursor of said larger database.

22. The apparatus according to Claim 16 characterized in that said plurality of pointers directly identify said second plurality of data objects.

23. The apparatus according to Claim 16 characterized in that said plurality of pointers comprise said second plurality of data objects.

24. The apparatus according to Claim 16 characterized in that said object oriented database manager further includes a stream element class of objects, and means for creating a second plurality of instances of said stream element class of objects, a respective one of said plurality of pointers identifying a respective one of said second plurality of

instances of said stream element class.

25. The apparatus according to Claim 24 characterized in that said second plurality of instances of said stream element class of objects includes a pointer for identifying a respective one of said second plurality of data objects.

26. The apparatus according to Claim 24 characterized in that said second plurality of instances of said stream element class of objects comprises a respective one of said second plurality of data objects.

27. The apparatus according to Claim 16 characterized in that said maximum attribute is selected to be less than or equal to the maximum number of pointers which may be stored in said data storage device.

28. The apparatus according to Claim 16 characterized in that said first method comprises:

means for deleting at least one of said plurality of pointers from said stream class attributes; and

means for inserting said pointer for identifying a data object immediately preceding the data object identified in said current attribute, into said stream class attributes.

29. The apparatus according to Claim 16 characterized in that said second method comprises:

means for deleting at least one said plurality of pointers from said stream class attributes; and

means for inserting said pointer for identifying a data object immediately succeeding the data object identified in said current attribute, into said stream class attributes.

30. The apparatus according to Claim 17 characterized in that said third method comprises:

means for deleting all of said plurality of pointers from said stream class attributes; and

means for inserting said pointer for identifying said first data object into said plurality of pointers.

31. The apparatus according to Claim 17 characterized in that said third method comprises:

means for deleting all of said plurality of pointers from said stream class attributes; and

means for inserting said plurality of pointers for identifying said stream class, with the inserted plurality of pointers beginning with said first data object.

32. The apparatus according to Claim 17 characterized in that said fourth method comprises:

means for deleting all of said plurality of pointers from said stream class attributes; and

means for inserting said plurality of pointers for identifying said stream class, with the inserted plurality of pointers ending with said last data object.

33. The apparatus according to Claim 16 characterized in that said second plurality of data objects comprises a second plurality of sequential data objects selected from said first plurality of data objects.

34. The apparatus according to Claim 16 characterized in that said stream class attributes further comprise:

a number attribute for identifying the number of said first plurality of objects to be read from said data storage device at one time.

35. The apparatus according to Claim 34 characterized in that said maximum attribute is an integer multiple of said number attribute.

36. The apparatus according to Claim 16 characterized in that said means for creating an instance comprises means for creating an instance of said stream class object for each user of said object oriented database management system.

37. The apparatus according to Claim 16 characterized in that said means for creating an instance comprises means for creating an instance of said stream class object for each query of said database in said object oriented database management system.

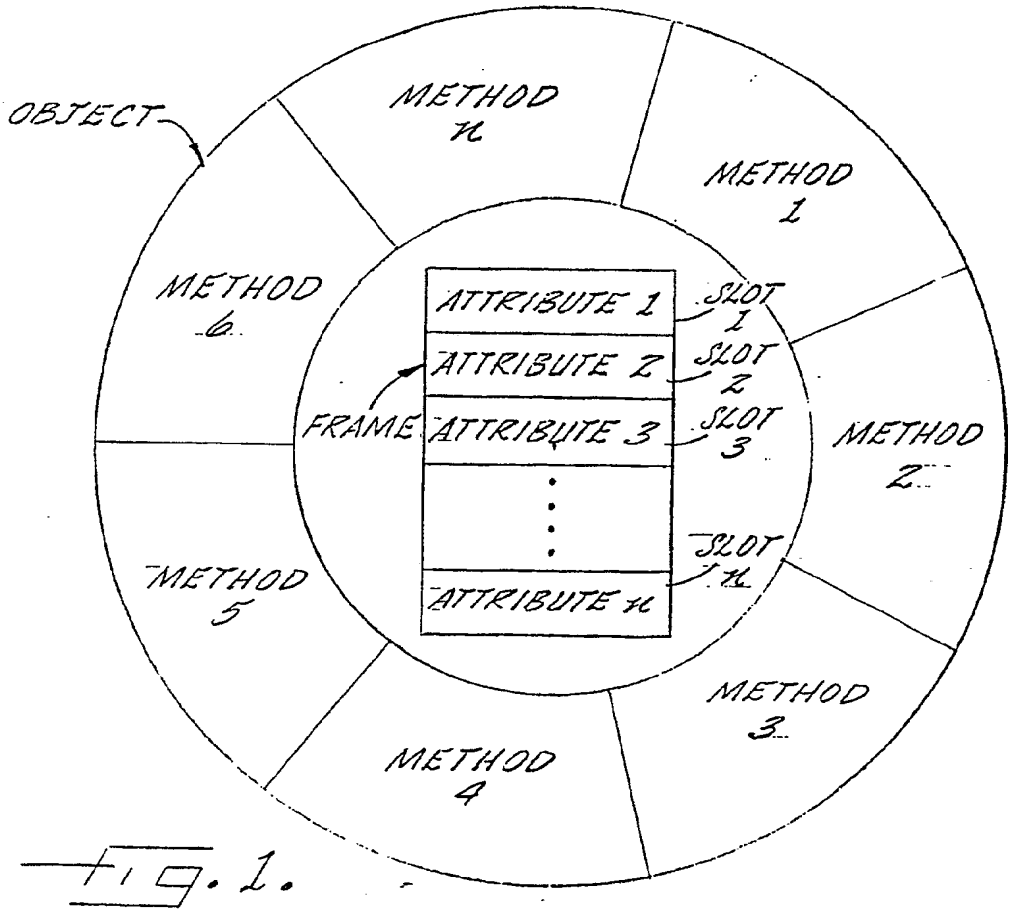

OBJECT
METHOD n
METHOD 1
METHOD 6
METHOD 2
METHOD 5
METHOD 3
METHOD 4
FRAME
ATTRIBUTE 1
ATTRIBUTE 2
ATTRIBUTE 3
ATTRIBUTE n
SLOT 1
SLOT 2
SLOT 3
SLOT n

Fig. 1.

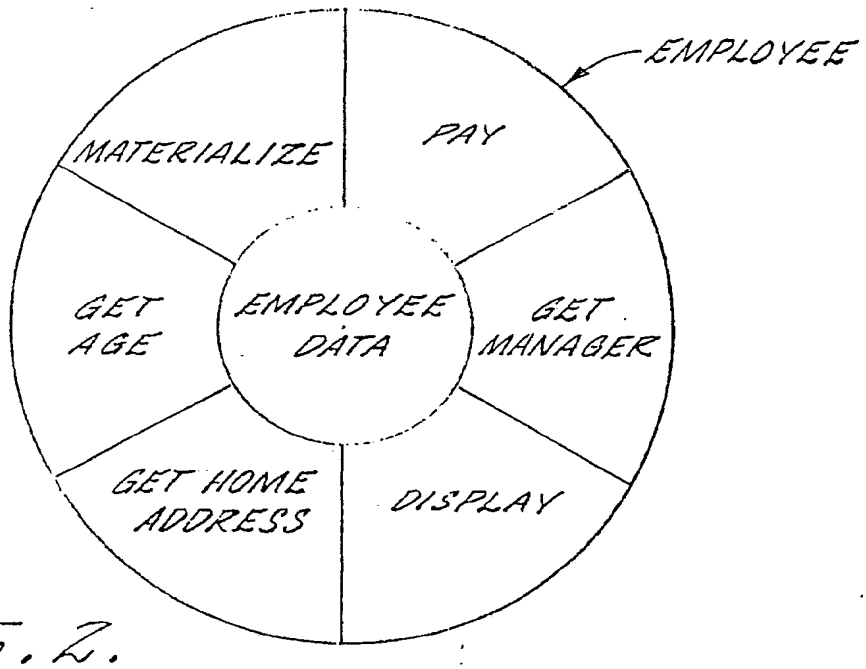

EMPLOYEE
MATERIALIZE
PAY
GET AGE
EMPLOYEE DATA
GET MANAGER
GET HOME ADDRESS
DISPLAY

Fig. 2.

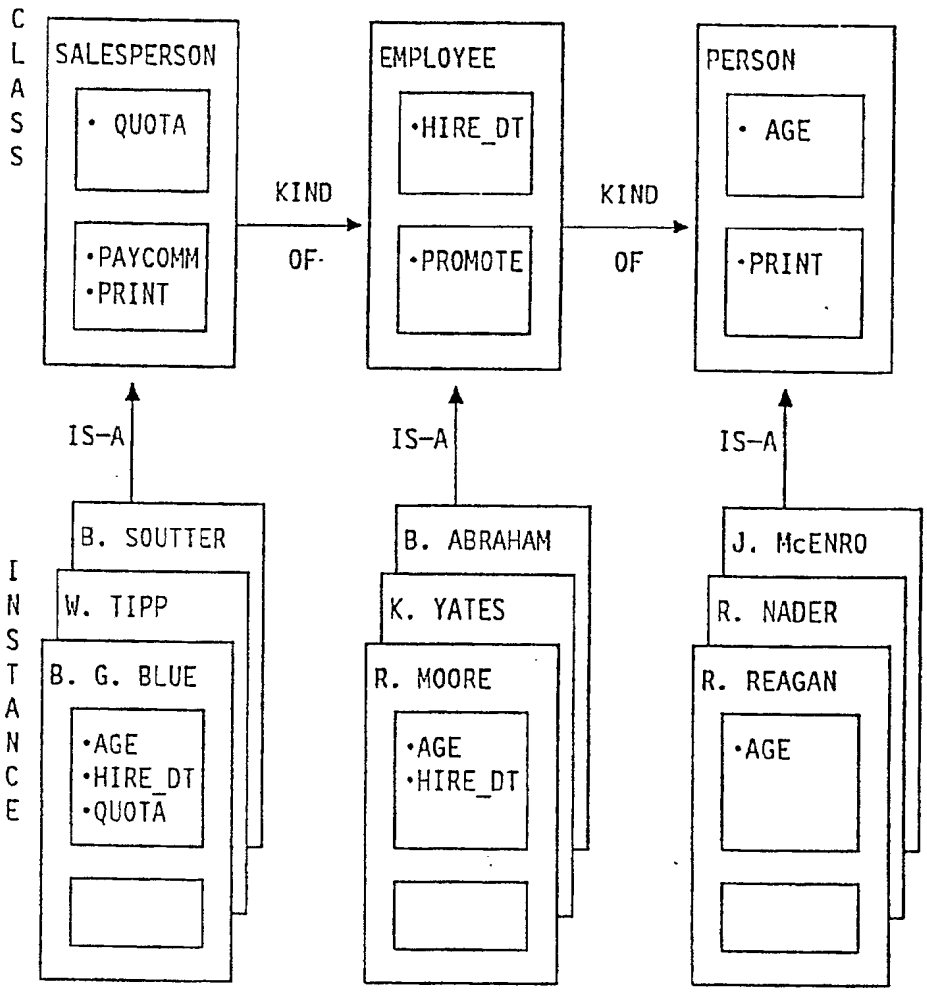

CLASS
SALESPERSON
• QUOTA
•PAYCOMM
•PRINT
KIND OF
EMPLOYEE
•HIRE_DT
•PROMOTE
KIND OF
PERSON
• AGE
•PRINT
IS-A
IS-A
IS-A
INSTANCE
B. SOUTTER
W. TIPP
B. G. BLUE
•AGE
•HIRE_DT
•QUOTA
B. ABRAHAM
K. YATES
R. MOORE
•AGE
•HIRE_DT
J. McENRO
R. NADER
R. REAGAN
•AGE

fig. 3.

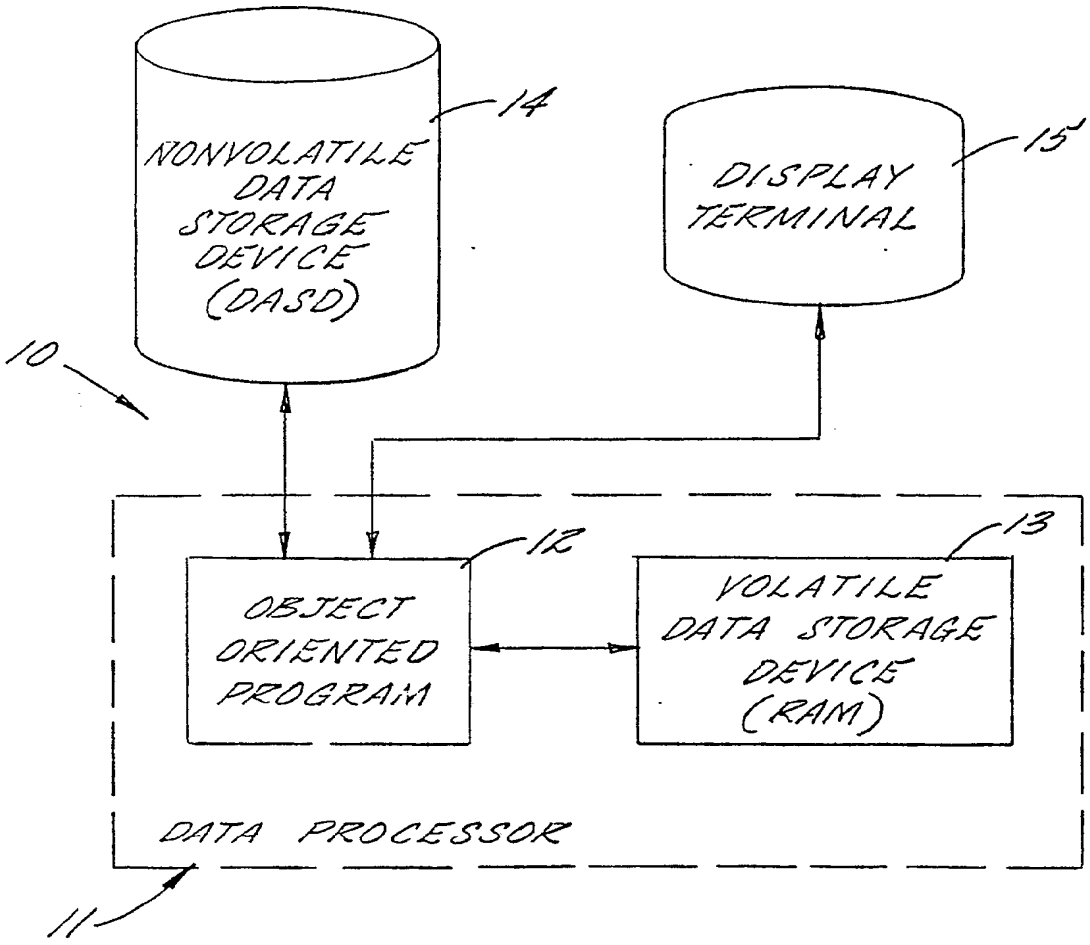

NONVOLATILE DATA STORAGE DEVICE (DASD)
14
DISPLAY TERMINAL
15
10
OBJECT ORIENTED PROGRAM
12
VOLATILE DATA STORAGE DEVICE (RAM)
13
DATA PROCESSOR
11

Fig. 4.

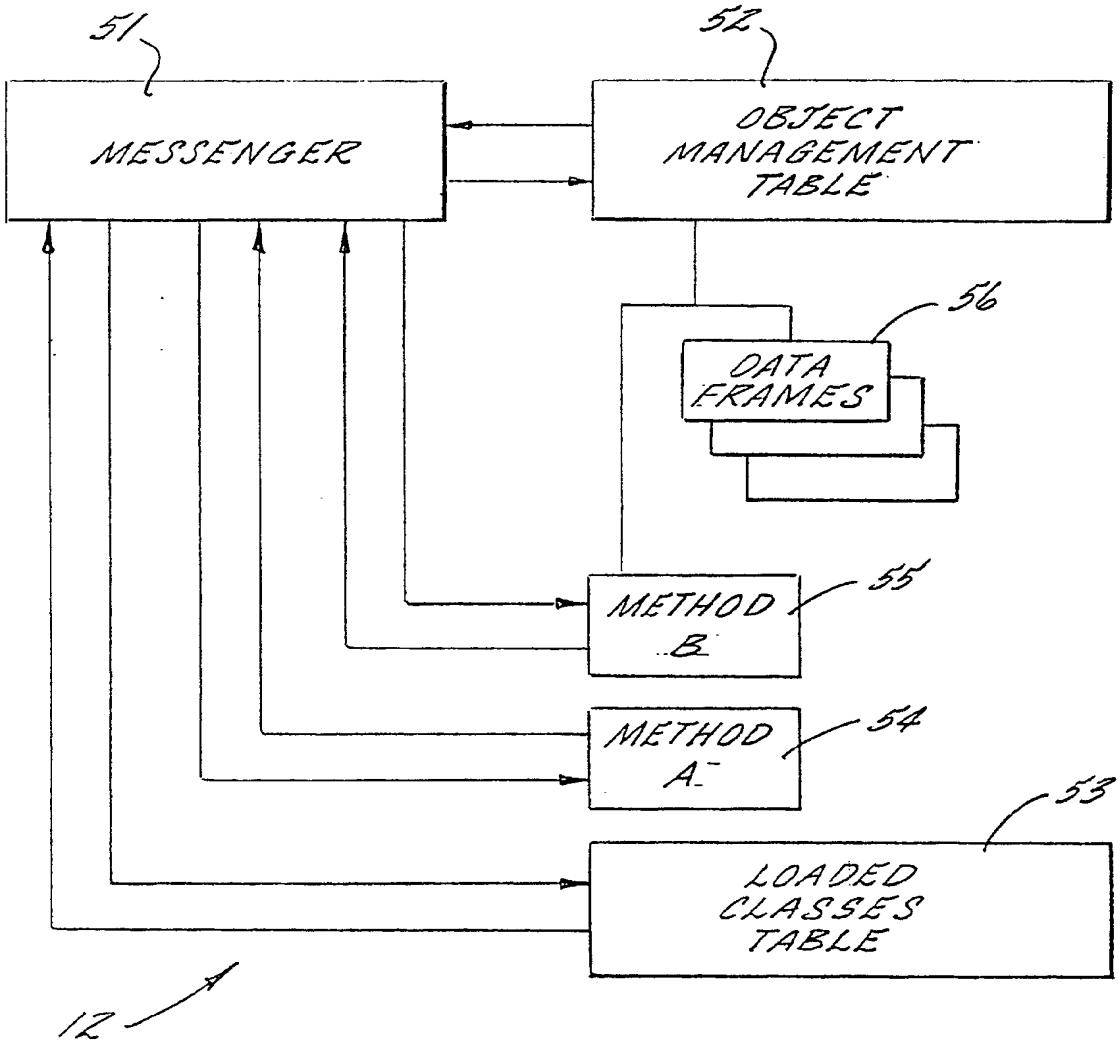

51
52
MESSENGER
OBJECT MANAGEMENT TABLE
56
DATA FRAMES
55
METHOD B
54
METHOD A
53
LOADED CLASSES TABLE
12

Fig. 5.

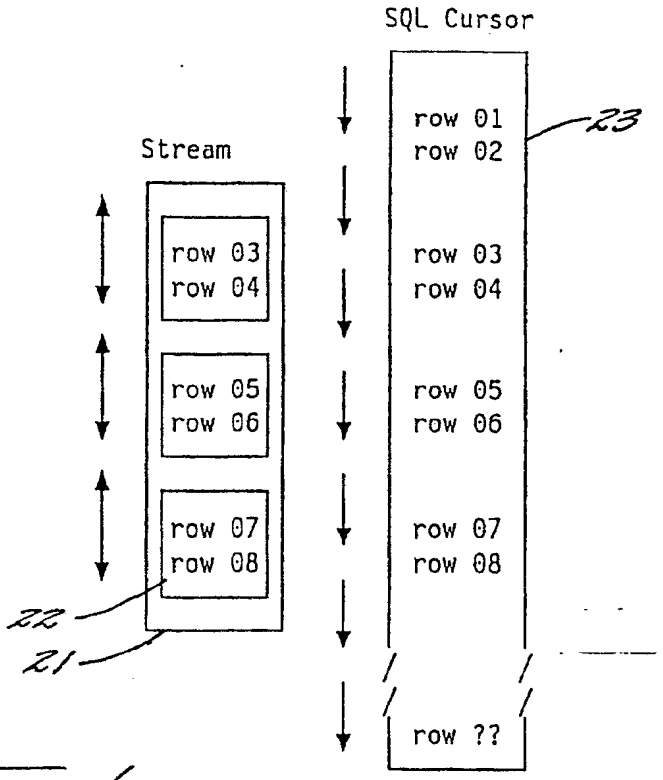

SQL Cursor
Stream
row 01
row 02
row 03
row 04
row 05
row 06
row 07
row 08
row ??
23
22
21

Fig. 6.

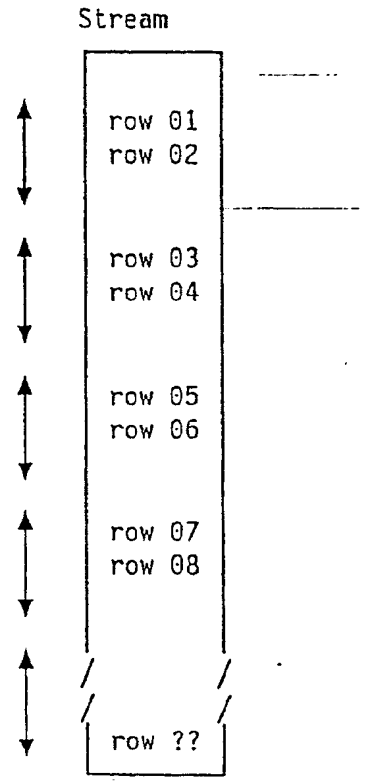

Stream
row 01
row 02
row 03
row 04
row 05
row 06
row 07
row 08
row ??

Fig. 7.

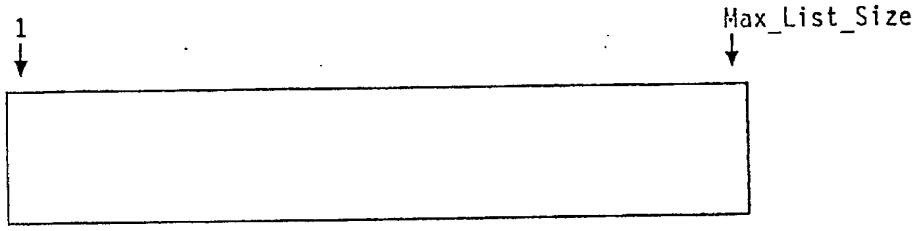



Fig. 8.

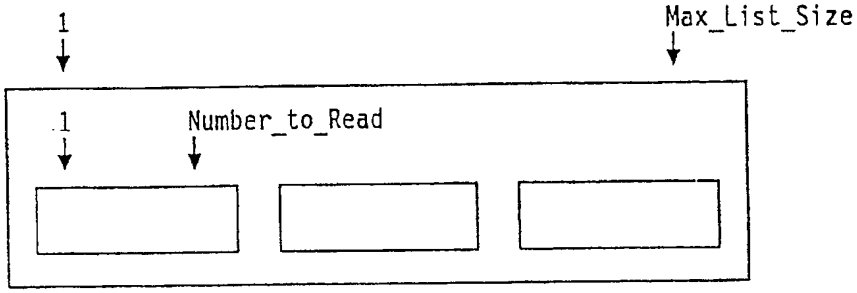



Fig. 9.

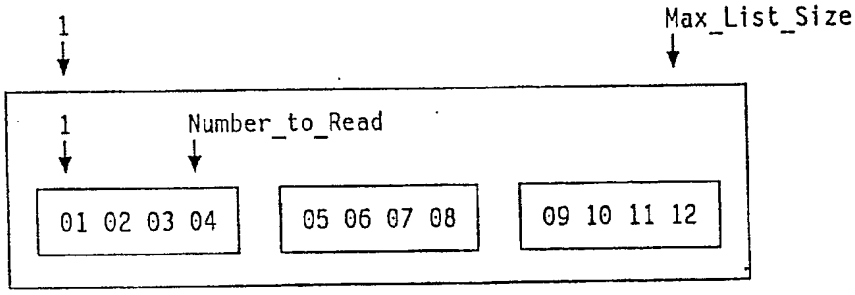



Fig. 10.

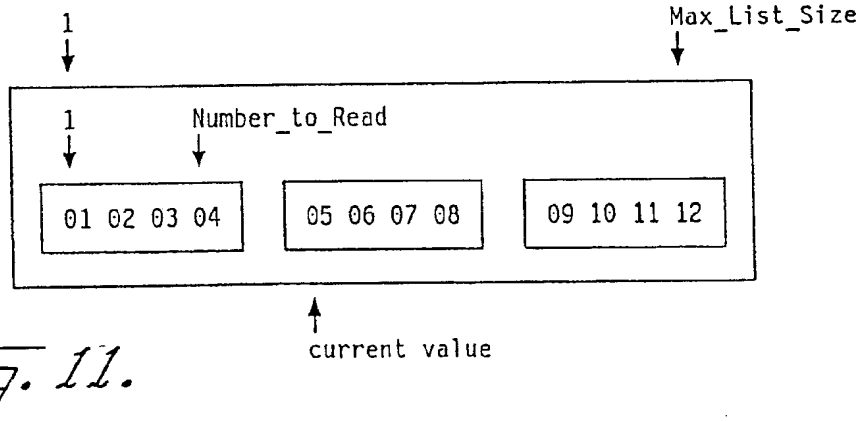



Fig. 11.

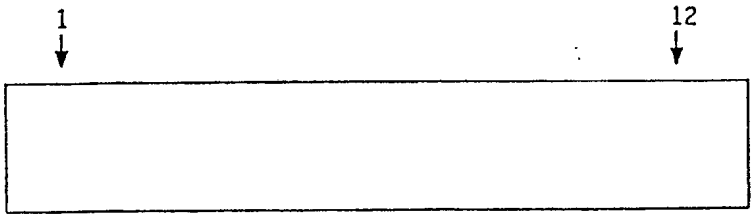



Fig. 12.

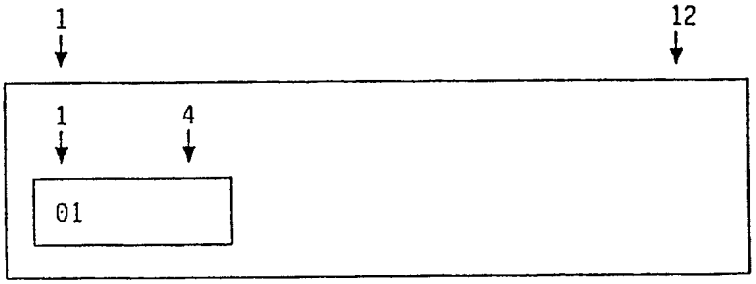



Fig. 13.

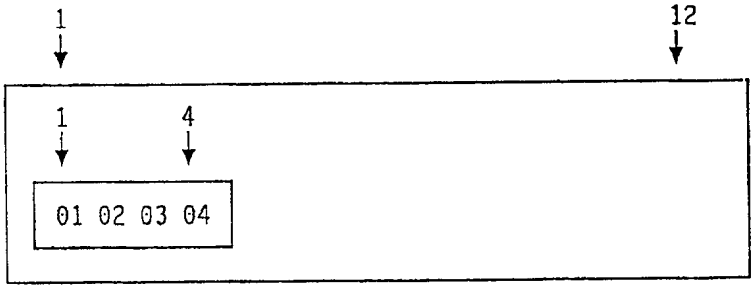



Fig. 14.

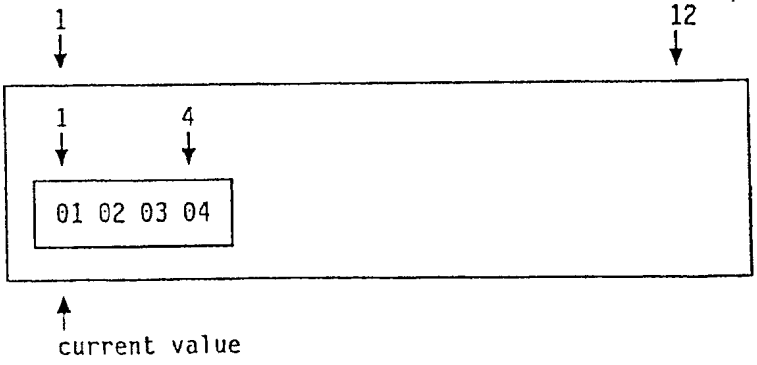



Fig. 15.

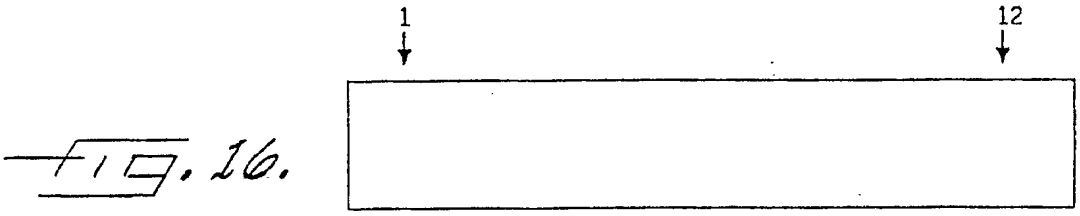



Fig. 16.

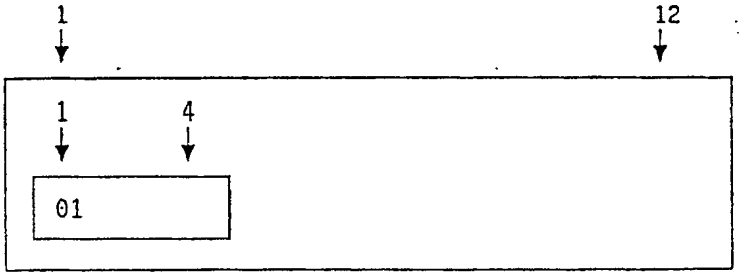



Fig. 17.

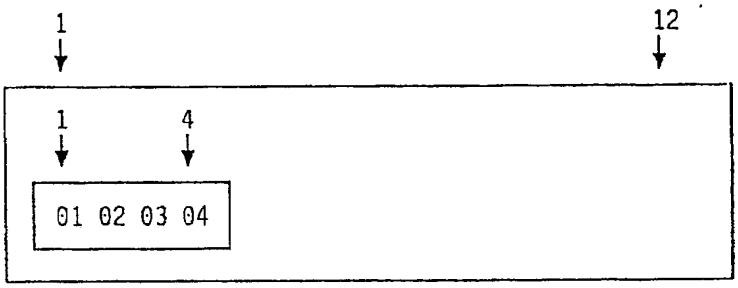



Fig. 18.

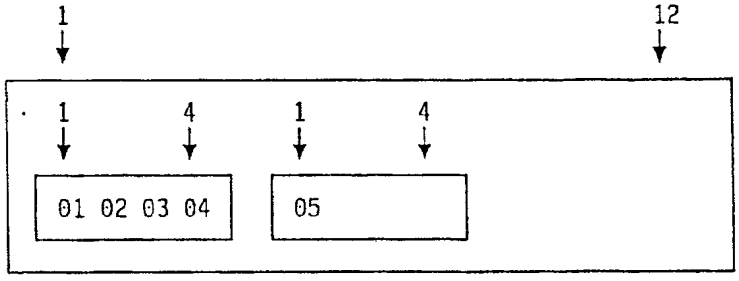



Fig. 19.

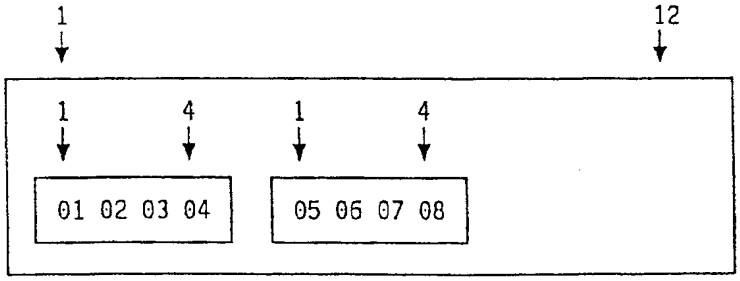



Fig. 20.

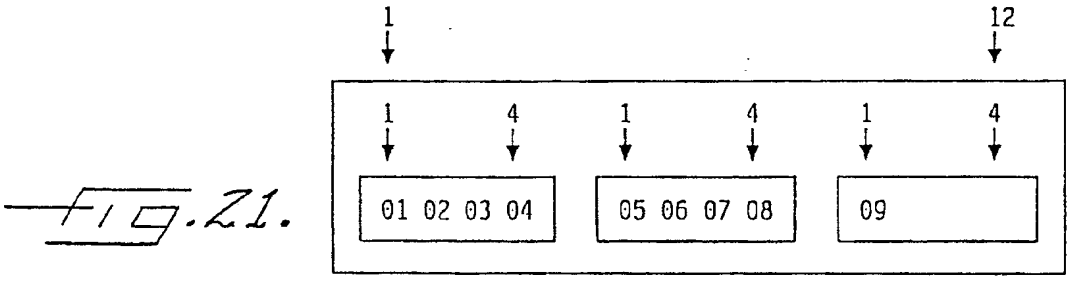



Fig. 21.

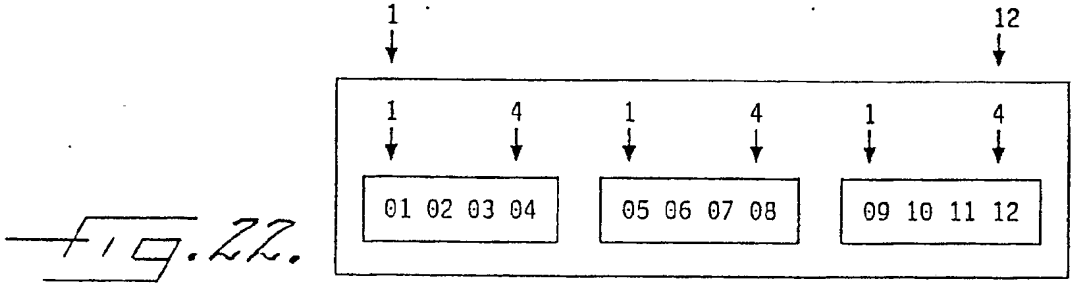



Fig. 22.

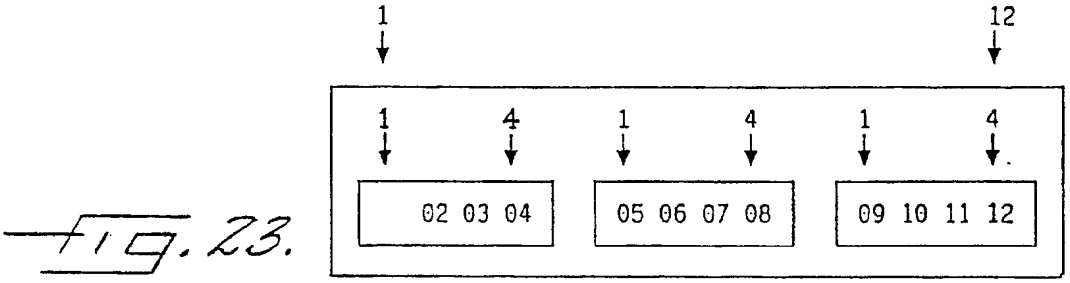



Fig. 23.

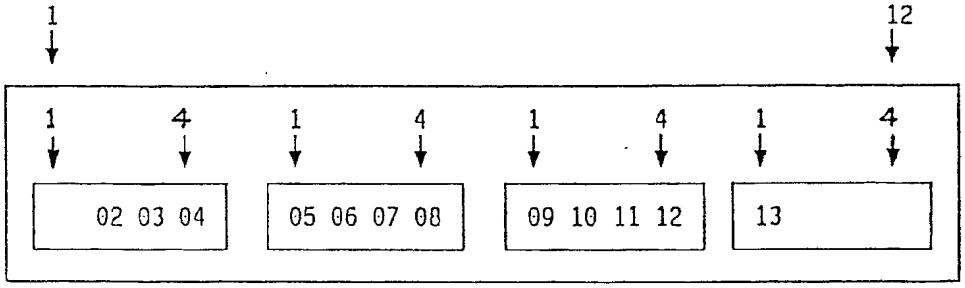



Fig. 24.

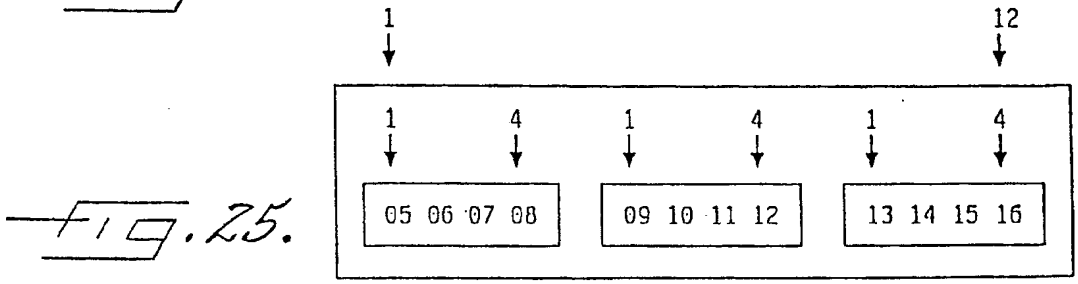



Fig. 25.

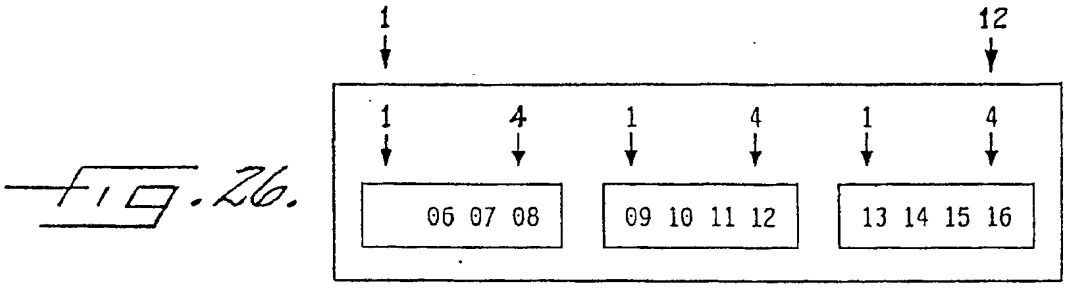



Fig. 26.

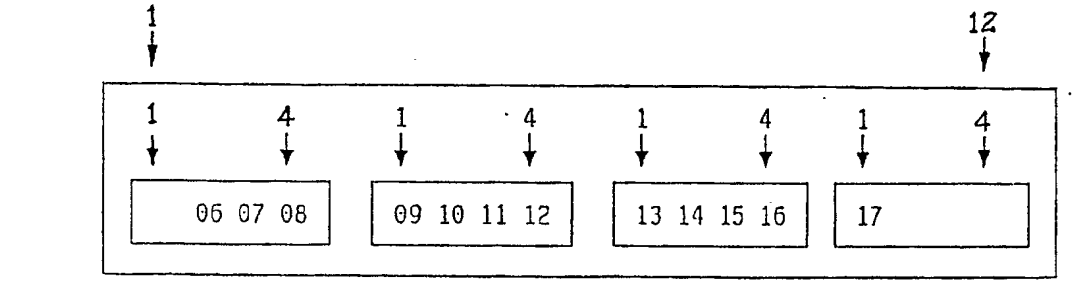



Fig. 27.

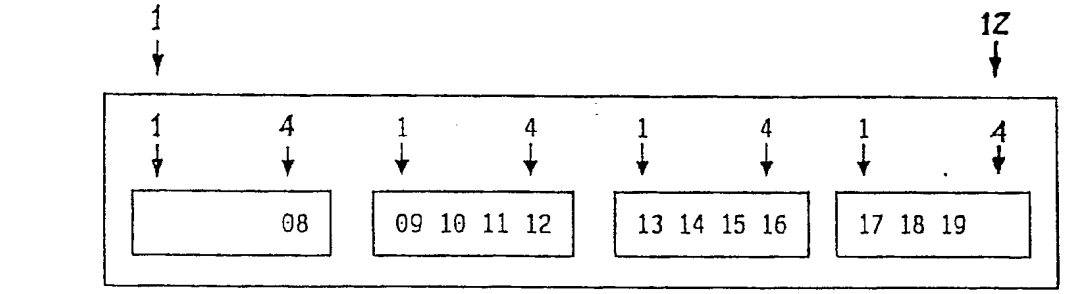



Fig. 28.

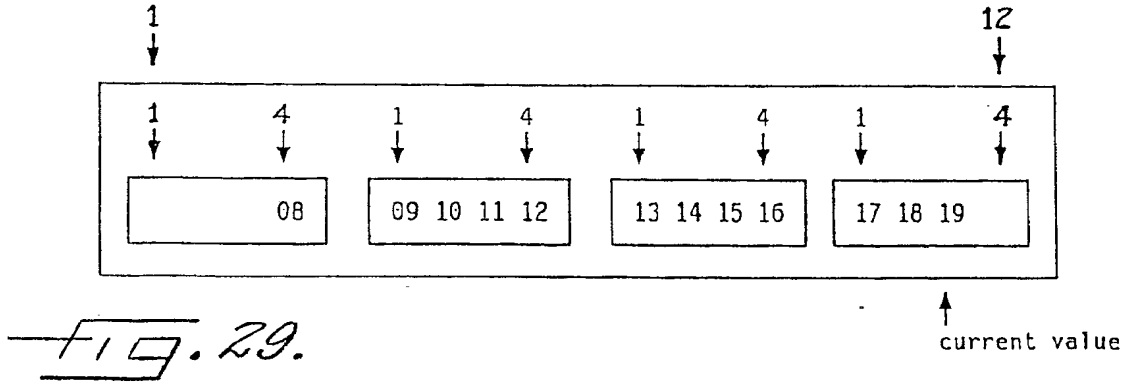



Fig. 29.

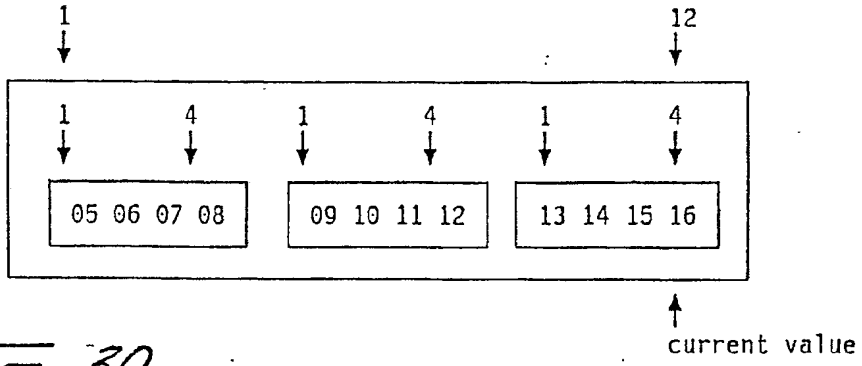



Fig. 30.

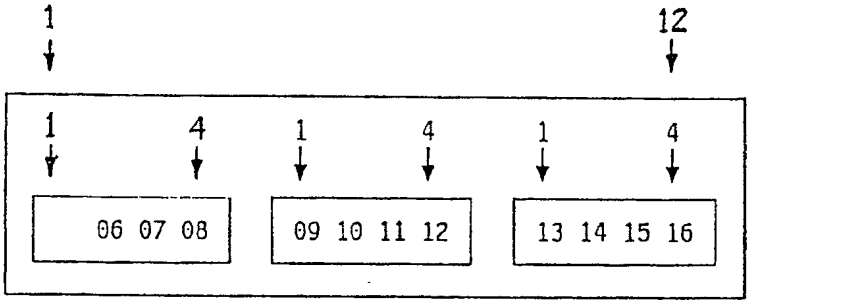



Fig. 31.

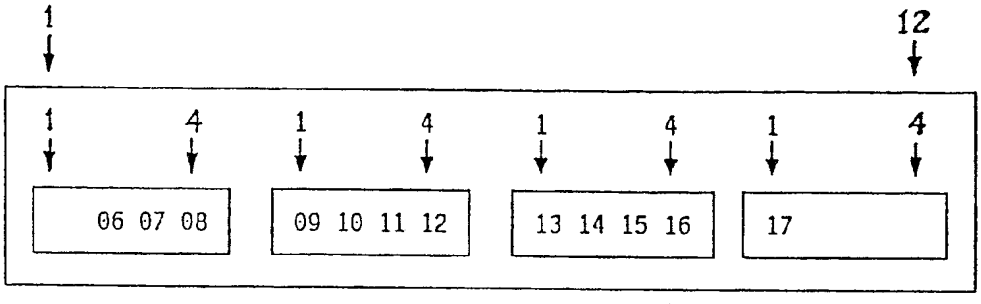



Fig. 32.

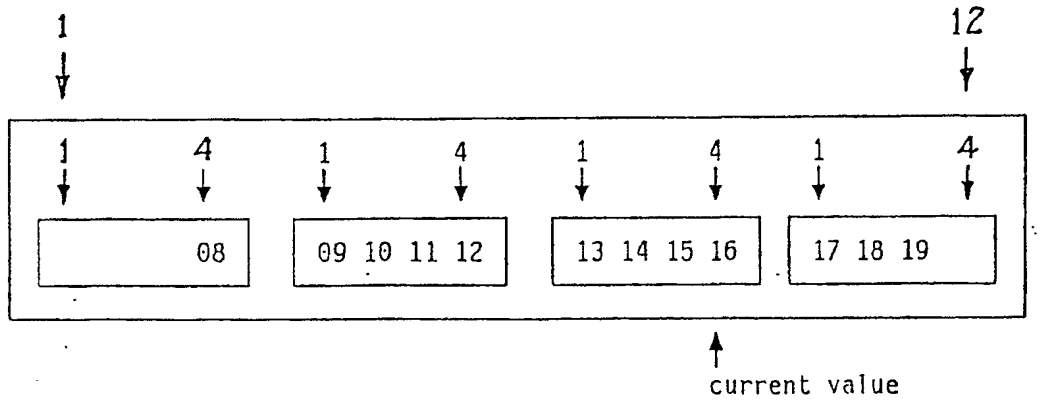

1
12
1
4
1
4
1
4
1
4
08
09 10 11 12
13 14 15 16
17 18 19
current value

Fig. 33.

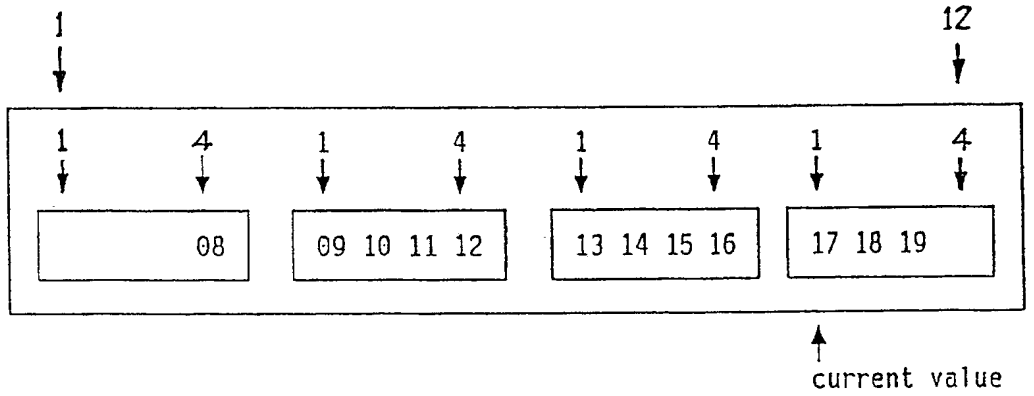

1
12
1
4
1
4
1
4
1
4
08
09 10 11 12
13 14 15 16
17 18 19
current value

Fig. 34.

Fig. 35.

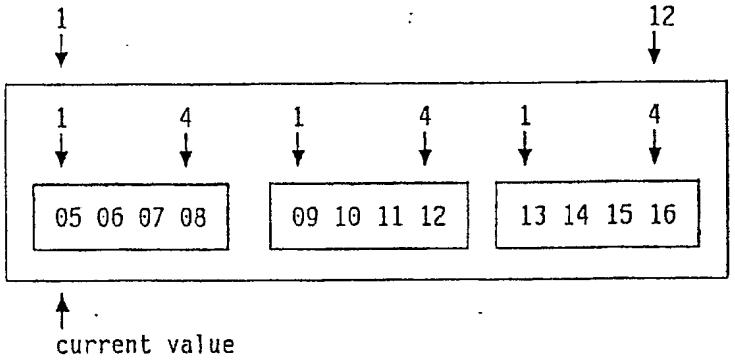



Fig. 36.

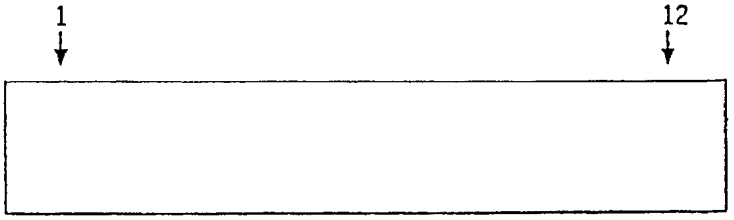



Fig. 37.

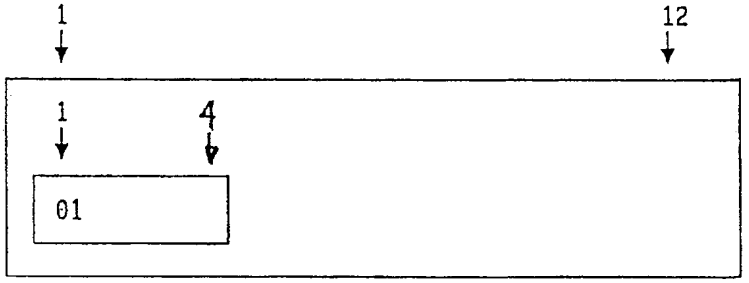



Fig. 38.

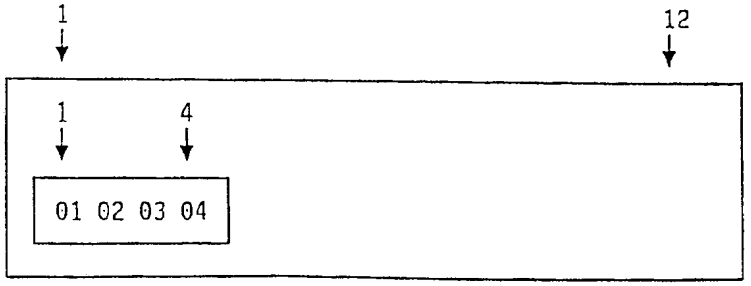


FIG. 39.

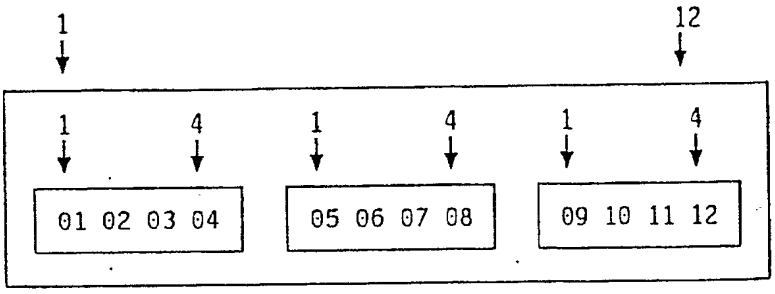
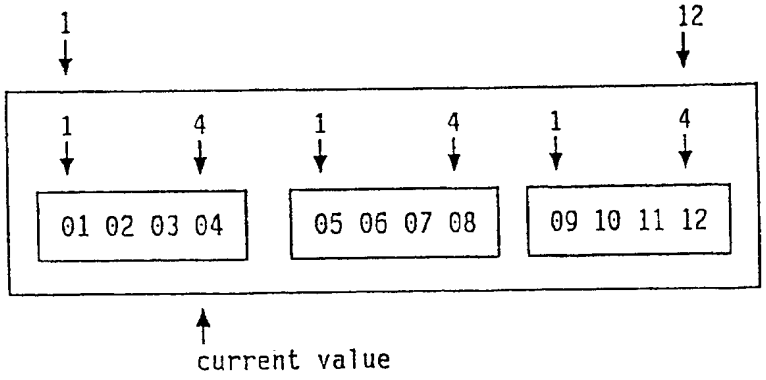

1
12
1
4
1
4
1
4
01 02 03 04
05 06 07 08
09 10 11 12

FIG. 40.


1
12
1
4
1
4
1
4
01 02 03 04
05 06 07 08
09 10 11 12
current value